Charon message-passing toolkit for scientific computations

Rob F. Van der Wijngaart
MRJ Technology Solutions
NASA Ames Research Center, Moffett Field, CA 94035

Abstract: The Charon toolkit for piecemeal development of high-efficiency parallel programs for scientific computing is described. The portable toolkit, callable from C and Fortran, provides flexible domain decompositions and high-level distributed constructs for easy translation of serial legacy code or design to distributed environments. Gradual tuning can subsequently be applied to obtain high performance, possibly by using explicit message passing. Charon also features general structured communications that support stencil-based computations with complex recurrences. Through the separation of partitioning and distribution, the toolkit can also be used for blocking of uni-processor code, and for debugging of parallel algorithms on serial machines. An elaborate review of recent parallelization aids is presented to highlight the need for a toolkit like Charon. Some performance results of parallelizing the NAS Parallel Benchmark SP program using Charon are given, showing good scalability.

1 Introduction

Writing message-passing codes is a tedious task for all but the simplest applications. Other parallelization tools are (becoming) available, such as compiler directives for (virtual) shared-memory systems, and parallelizing compilers like SUIF [27], and the commercial products KAP and FORGE Explorer, but these are usually not up to the task of large scale parallelization. High Performance Fortran (HPF) [33] holds the promise of efficient parallelization of certain classes of problems, but the language lacks expressivity in general (see Section 2), and writing efficient HPF compilers has proven to be a daunting task (for a recent evaluation of HPF compilers, see [38]). For the majority of large-scale scientific applications message passing is still the method of choice.

The advantages of message passing are clear. The user has complete control over exploitation of concurrency and distribution of data. The separate processor address spaces and the explicit message passing calls provide a simple programming model. They also enable tuning, since the often costly communications are completely managed by the programmer. The major disadvantage, contrary to common belief, is *not* the bookkeeping associated with the placement of message-passing calls. In a typical application the fraction of lines of program text involving communications is small. What makes message passing truly cumbersome in most scientific computing programs is the explicit management of the domain decomposition, i.e. the restriction of data structures and operations to individual processors[1]. The processor sees only a small 'window' of the entire distributed data structure. Moreover, message-passing programs cannot be developed gradually from a serial code. Domain decompositions are all-pervasive, and the entire program must be converted all at once. This puts message-passing at a distinct disadvantage compared to the shared-memory paradigm,

[1] We speak mostly of *processors* in this paper, although sometimes *processes* would be more appropriate.

which allows piecemeal conversion of legacy code through parallelizing directives.

Charon offers a vehicle for easy development of efficient message-passing programs. It is a toolkit that aids engineers in parallelizing scientific programs for structured-grid applications. Both legacy code conversion and development from scratch are supported. Charon provides a small library of functions that create, manipulate, and interrogate domain decompositions and the distributed variables defined on them. Charon is also portable, requiring no special operating system or hardware support; it is programmed in ANSI C, is callable from Fortran and C, and is built on the *de facto* message-passing standard MPI [45].

Emphasis within this project is on rapid program development and debugging, and subsequent piecemeal performance tuning. To support this approach, functions that manipulate

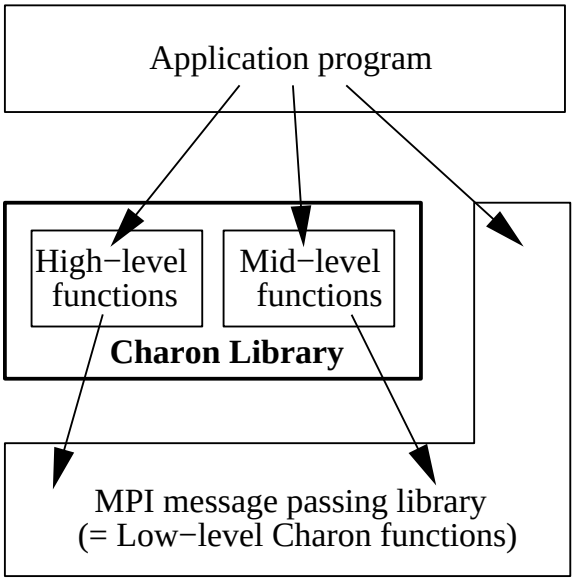

Figure 1: Charon software structure

distributed variables are provided at three levels of abstraction, as depicted in Figure 1. The highest level is the simplest to use, but also the least efficient. It is a device that emulates serial program execution on distributed data. Salient features are a simulated single program counter, application of the owner-computes rule, and automatic synchronization. Strict sequential consistency in the sense of Lamport [36] is guaranteed. The sole purpose is to distribute the data without any change in program structure. Invocation of the Charon library at this level is the first stage in the parallelization of a user program.

The intermediate level is still easy to use, but more efficient. It consists of a collection of communication routines, augmented with functions that control granularity, synchronization, and concurrency, and that allow relaxation of the owner-computes rule. Indexing of array variables is global, so the image of (distributed) shared memory is retained.

The lowest level of abstraction involves local indexing of variables, i.e. direct local memory access without library intervention, and (possibly) explicit message passing; it offers the highest efficiency. Calls at all three levels can be freely mixed, allowing a gradual transition from low to high efficiency and a peaceful coexistence of paradigms.

Charon is intended for *difficult* structured-grid problems. It offers support for numerical

methods that involve recurrences that cannot be expressed in a data-parallel way. It also provides very general, possibly dynamic, domain decompositions. Problems that do not need this flexibility—most notably explicit methods, which exhibit natural data parallelism—may be more easily solved using some of the systems described in Section 2, such as KeLP [19], OVERTURE [9], or PETSc [5]. Yet even for such problems the Charon approach offers advantages, because of the control the user has over data lay-out (e.g. array padding or index interchange to improve cache utilization, reuse of workspace) and communications.

The remainder of this paper is organized as follows. In Section 2 an extensive overview is given of previously published parallel programming paradigms and development environments. Their limitations are discussed, from which follows the need for a library like Charon. Section 3 presents the philosophy behind the Charon design. In Section 4 we describe the Charon data distribution and manipulation facilities. Subsequently, in Section 5, we explain the library functions that provide parallel execution support at the highest level of abstraction. In Section 6 we show how to apply the distribution and high- and mid-level execution support functions to distribute and parallelize programs. For this we choose a very simple example, programmed in C, based on the NAS Parallel Benchmarks (NPB) Block Tri-diagonal problem, and also the full NPB Scalar Penta-diagonal benchmark program [4], coded in Fortran. For the latter we also show actual performance results. We conclude this section with practical tips for easy code conversion. Finally, in Section 7, we outline the work remaining to complete the library.

2 Survey of related projects

We restrict our investigation to scientific-computing problems that are best solved using domain decomposition. We specifically target the solution of multi-dimensional-array based problems on distributed-memory multi-computers. There are several ways in which parallelization aids can be classified. Here we adopt the following two simple categorizations:

- threads, tasks, or (virtual) processes can be either *named* or *nameless*. A task is nameless if it cannot ask for its ID. If the tasks are named, *communications* between them can be *implicit* or *explicit*.
- the *domain decomposition* can be *implicit* or *explicit*. If the domain decomposition is explicit, there is the possibility (but not the necessity) that the *physical data distribution* is also explicitly known.

If tasks are nameless and the domain decomposition is implicit, all the user can do is indicate which program segments can be executed in parallel; interaction is limited to giving hints regarding concurrency. Spawning and joining of parallel tasks is performed by the system, often on a loop-by-loop basis. Since the user does not know which statements are executed by which processor, nor where data resides, this kind of parallelization aid gives an image of shared memory. Vectorization and multi-tasking compiler directives defined by Cray Research [55], and parallelization directives defined by Advanced Parallel Research (FORGE) [54], Silicon Graphics (MIPSpro Power Fortran (X3H5 compliant) and C (pragma-based directives)), and in the draft report by the defunct Parallel Computing Forum X3H5 committee fit this description. These directives have the form of structured comments, and are ignored by non-parallelizing compilers. If no directives are given, some parallelization or

vectorization may still occur if the source code is simple enough to be analyzed by the compiler. Depending on the level of sophistication of the compiler, interprocedural dependency analysis may discover coarse-grain parallelism [54, 27]. Completely automatic parallelization is obtained by compilers such as SUIF [27] and KAP by Kuck and Associates.

The advantage of these tools is that they allow quick parallelization of legacy codes. Either nothing needs to be done at all, or only some structured comments are inserted. However, the disadvantages are several. In the case of a non-uniform memory access (NUMA) computer, the lack of control over data placement can lead to severe performance degradation. In many scientific computing programs there is not a single optimal data distribution for the entire code. A user can select a reasonable compromise. But an automatically generated decomposition can be extremely poor for certain parts of the program, leading to frequent remote memory accesses and/or page migrations. The problem with nameless tasks is insufficient expressibility of parallelism. At best, the user can specify that a program structure—typically a loop—can be executed in parallel (e.g. X3H5's `C$PAR PARALLEL DO`). At worst, there is no control at all, and the compiler may extract parallelism at the wrong level of granularity. Interprocedural analysis may help, but has its limitations, since not all dependencies can be resolved at compile time [26]. Regarding the lack of expressivity of parallelizing compilers, one of the most useful control structures in parallel programming, the pipeline, cannot be expressed without named tasks. The only directives allowed indicate data parallelism.

In certain Distributed Shared Memory (DSM) systems concurrent task ID's are available, either through intrinsic function calls, or through parameters passed by the user when spawning a task. Examples are Treadmarks [2], Cilk [52], Cashmere [46], Munin [12], Ivy [37], and Shasta [42]. The recently announced OPEN-MP standard, endorsed by a number of vendors, tries to unify existing DSM application programmer interfaces. The main difference between the DSMs is in the type of memory consistency offered, and in the support for hardware shared memory within a single node of an interconnection network (Shasta and Cashmere). While it is possible to code complex parallel constructs using the task ID, the lack of information on where data actually resides puts performance in jeopardy. In some systems pages are assigned to a 'home' processor by the system once and for all, while in others some limited redistribution may take place. This may cause mismatches between data placement and task execution in the case of nontrivial and/or dynamic data dependencies.

But making the domain decomposition explicit is not always sufficient. HPF [33] and the related projects HPC++ [32], Vienna Fortran [14], C* [35, pp. 450–459], Annai [16], CM Fortran [53], PC++/Sage++ [23], Fortran D [22] (augmented with the CHAOS runtime support procedures [40]), Mentat [24], the explicit parallel Fortran syntax bindings from the draft X3H5 document, etc., allow the user to specify or to suggest how to distribute the data. But they do not provide control structures to express parallelism explicitly beyond the `PARALLEL DO` (X3H5), `FORALL` and `INDEPENDENT` (HPF), or equivalent constructs. This is because the tasks are nameless; even though a group of (virtual) tasks exists, an individual task cannot ask for its name.

Vienna Fortran, Annai and Fortran D fix some of the deficiencies of HPF, for example by making more explicit which (virtual) processor executes what set of statements in a parallel loop (through the `ON` clause in Fortran D and Vienna Fortran), by providing more general data distributions (through the `BLOCK_GENERAL` distribution in Annai's Parallelization Sup-

port Tool (PST), and through user-defined mappings in Fortran D, Vienna Fortran, and PST), by defaulting to private, rather than shared data structures (Fortran D, Annai), and by offering reductions in parallel loops within the language, rather than through the awkward `EXTRINSIC(HPF_LOCAL)` mechanism. But since these languages provide only implicit (compiler-generated) communications, granularity is often unnecessarily fine, unless hand tuning is applied [28]. Moreover, the programs are strictly SPMD, and shared data types are distributed over the entire processor set. SAGE, Mentat and HPC++ have the added disadvantage that they require the use of C++, which is not the language of choice for most numerical analysts. HPF, CM Fortran, C* and some other systems define virtual processors, which are mapped to physical processors by the compiler. While this sometimes constitutes a programming convenience, preventing the user from identifying processors for target data distribution can severely degrade performance.

The problem with most of these parallelization tools is that they do not provide a mechanism to express task parallelism. All processors may 'attack' a `FORALL` construct in HPF, but there is no way of assigning some processors to a certain task, while others proceed with another. Only data parallelism can be expressed explicitly. The reason for this is that explicit task parallelism requires placement by the user of communications between *interdependent* tasks, something that tool writers have traditionally tried to avoid. All task parallelism recognized by parallelizing compilers—and there are several that are capable of extracting some—is implicit, and the communications are inferred. As an example we mention the pipeline feature in the Fortran D compiler, which usually produces inefficient code [28], because of the difficulty of automatically choosing the proper pipeline grouping factor.

Another package capable of resolving recurrences through pipelines is CAPTools [31], which uses a dialogue with the user to parallelize legacy codes. CAPTools—like Fortran D and some of the proprietary and public domain HPF (pre-)compilers, for example the ADAPTOR/Bouclettes system [8]—automatically detects certain common data dependencies and inserts the proper control structures and message passing calls. The result is a translated source text that can be edited for further tuning. But CAPTools still has some of the limitations of most other parallelizing tools. Data distributions are essentially the same as those of HPF (but without redistribution capabilities), the programming model is strictly SPMD, and the quality of the parallel code produced depends on the capability of the underlying dependency analysis engine to recognize complex data dependencies, as well as on the judicious selection of the proper granularity. Forge Explorer [54] resembles CAPTools in its capability to parallelize programs once the user has indicated interactively how data should be distributed, but is more restricted in its domain decompositions (only one array dimension may be partitioned), and, again, only data parallelism is supported.

Attempts to provide an expression mechanism for task parallelism include Fx [25], Shared Data Abstractions (SDA's) [13], Sisal [10], Fortran M [20], Split C [18], Linda [11], and the HPF/MPI bindings defined by Foster et al. [21]. Split C provides complete expressibility of task parallelism, but poor facilities for data distribution and little user support for manipulation of shared data types (termed *spread arrays*). Fx and Fortran M, and to a lesser extent SDA, the HPF/MPI bindings, Sisal, and Linda, are aimed at multi-disciplinary applications, where certain disjoint tasks can be run concurrently. They set up explicit communications mechanism (input and output mapping directives in Fx, MPI messages between distributed objects in HPF/MPI, channels and ports in Fortran M, stacks in SDA, and objects in tu-

ple space in Linda) that can transfer information between these tasks. Fx, Fortran M and HPF/MPI do so in a tightly coupled fashion, whereas Linda and SDA do not link sender and receiver directly. Of those investigated here, Sisal is the only purely functional language. It contains no explicit expression mechanism for concurrency. Rather, Sisal relies on the guaranteed absence of side effects in function calls and the language distinction between serial and independent (data parallel) loops to derive parallel code.

The problem with the above approaches is that there is no good support for global (shared) data types within the tasks. Just as in the 'bare' message-passing environment, the user is responsible for interpreting the meaning of the parts of distributed data structures (mostly arrays) owned by the individual processors when using Fx, Fortran M or Linda (tags in Linda can help alleviate this problem, but creating and maintaining those is still the programmer's responsibility). SDA, Sisal and HPF/MPI do support data types globally known to (sets of) tasks, but allow only data parallel or serialized operations on such data types.

Apparently, the user must choose between either task parallelism and private, non-shared data types, or data parallelism and shared data types. What we want is shared data types for programming convenience—where sharing may be among a subset of the processors in the system—and task parallelism for flexibility. One way of accomplishing this is encapsulation through parallel libraries; tasks are issued as parallel, atomic tasks on globally defined, shared data types, and the library implementation, which may involve task parallelism, is hidden from the user. Opting for libraries represents a compromise, since no library can be completely general-purpose. The art of the library designer consists of choosing a system that is small enough that it can be mastered fairly easily, and large and flexible enough to solve more than the particular problem for which it was invented. Some of the interesting projects in this area are ScaLAPACK [7], KeLP [19], PETSc [5], OVERTURE [9], Global Arrays toolkit [39], //ELLPACK [29], PINEAPL [34], PARTI [47], DAME [17].

ScaLAPACK, part of the larger project "A Scalable Library for Numerical Linear Algebra," is a distributed-memory version of the linear-algebra library LAPACK. It allows matrix distributions that are a subset of the HPF array distributions, namely two-dimensional block-cyclic distributions. The library is built on top of lower-level serial (BLAS; Basic Linear Algebra Subprograms) and parallel (PBLAS; Parallel BLAS, and BLACS; Basic Linear Algebra Communication Subprograms) libraries. The efficiency of the library derives from the implementation of the fairly small set of machine-dependent routines of PBLAS and BLACS. Those problems that can be cast completely as numerical linear-algebra problems can be solved readily using ScaLAPACK, with almost the some ease—and a very similar user interface—as in LAPACK.

Some other linear-algebra libraries available or currently being developed are: JTpack90 [48], Aztec [30], LINSOL [44], PLAPACK [1], PSPARSLIB [41], and the The Global Arrays toolkit [39]. The latter targets dense matrix operations. It is different from all the other libraries in that it allows both task parallelism and data parallelism, a property that Charon also seeks to provide. Matrices are created and distributed in collective operations, but methods that operate on the matrices can be collective (in the vein of PBLAS [7] operations) as well as private (e.g. processor 1 may fetch a submatrix and perform an operation on it, with all other processors idling, or engaging in other tasks). However, distributions are limited to those in HPF, and are restricted to two-dimensional arrays.

Unfortunately, numerical linear algebra problems are often embedded in larger applications, which may lead to distribution incompatibilities. Moreover, most structured-grid applications do not construct system matrices explicitly, so they do not benefit from ScaLAPACK parallelization.

In addition to general-purpose linear-algebra libraries, a plethora of special-purpose parallel packages are being developed, some of which utilize the above linear-algebra libraries. They derive their utility from their execution efficiency, combined with their ease of use. However, most such libraries have specialized considerably, sacrificing generality and expandability for efficiency and simplicity.

- The Kernel Lattice Parallelism (KeLP [19]) project offers a convenient user interface for the solution of partial differential equations using structured, adaptively-refined grids. It provides functions for the creation, movement, reshaping and destruction of the refinements. However, grids and refinement are limited to aligned, Cartesian blocks. More seriously, KELP only provides coarse-level parallelism, and does not allow individual blocks to be further distributed among processors. Numerical operations performed on blocks must be data-parallel. This restricts numerical methods to explicit schemes, Jacobi- or colored-Gauss-Seidel-type point-relaxation, or Krylov-based solvers, all of which compute updates (or residuals, in the case of Krylov subspace methods) on a pointwise basis.

- The University of New Hampshire C* compiler [15] offers support for stencil computations on structured grids. Staging communications with neighboring processors in the case of so-called box-shaped difference stencils (see below), as described, for example, by Scherr [43], reduces the latency in massively parallel computations. However, since the support is provided within the context of the C* language [35, pp. 450–459], computations can only be performed in a data-parallel fashion.

- OVERTURE [9] is a C++ library for solving partial differential equations on serial and parallel computers. It provides a high-level specification and solution mechanism for partial differential equations on (collections of overlapping) structured grids, with provisions for adaptive refinement. OVERTURE contains procedures for stencil operations and a library of boundary conditions and integrators. Again, all operations are specified as data-parallel grid functions.

- //ELLPACK [29] offers three mechanisms for the parallel solution of PDE problems. Two are based on legacy code coupling and suffer from serial bottlenecks. The third approach is the only truly parallel problem solver in //ELLPACK, but it is limited to templates describing elliptic equations.

- The Parallel Automated Runtime Toolkit (PARTI [47]) at ICASE consists of two parts, PARTI proper, and multiblock PARTI. PARTI provides an interface to manipulate data structures related to unstructured meshes and general sparse matrices. It employs the celebrated inspector/executor model, in which loops over irregular data structures are preprocessed to determine their remote data requirements, and the pertinent communications and calls to gather/scatter routines are automatically inserted. This model

assumes processors can execute their own segments of loops independently once remote data is fetched. This restricts numerical methods to those expressible as data parallel loops.
Multiblock PARTI accommodates sets of interfacing structured grids ("blocks"). Blocks are assigned to sets of processors, and are updated independently, after which irregular communication takes place to update interface values. Within blocks fine-grain parallelism may be exploited through the use of Fortran-D-conforming data distributions and loops. Thus, multiblock PARTI extends the use of Fortran D by allowing task parallelism among the various blocks, but is restricted to the data parallelism expressible in Fortran D within individual blocks.

- DAME (DAta Migration Environment [17]) creates a homogeneous distributed virtual machine with a regular virtual topology to the application programmer, hiding the details of the irregularly connected, temporally and architecturally heterogeneous environment on which the application is actually run. Like the Global Arrays project [39], it targets dense matrix operations. DAME also offers explicit index conversion functions that translate global indices into local ones, and functions that extract from a specified data domain the part contained in the address space of the calling node. However, operations on distributed data sets are restricted to data parallel functions, and distributions are restricted to 2-dimensional block decompositions.

- The Parallel Industrial NumErical Applications and Portable Libraries (PINEAPL) project [34], has created the NAG Parallel Library, which provides an extension of the traditional NAG (Numerical Analysis Group) Fortran 77, Fortran 90, and C libraries. Support for the solution of partial differential equations (PDEs) consists of templates for the specification of (unstructured) grid and equation to be solved. Communication, which is shielded from the user, is based mainly on the BLACS [7] routines to ensure portability and efficiency. While reported scalability is good [34], functionality is rather limited due to the template nature of the library. In the PDE area only scalar Helmholtz and Poisson solvers are provided at present.

- The Portable, Extensible Toolkit for Scientific computation (PETSc [5]) is the most extensive and versatile of the parallelization support packages available today. Rather than providing a (necessarily restricted) template for the parallel formulation and solution of PDEs, it offers a set of functions for the creation, manipulation and destruction of high level distributed data types, such as vectors and matrices, and a collection of general-purpose linear and nonlinear equation solvers.
  Like in the Global Arrays [39] project, distributed data types are created collectively, but may be manipulated collectively (using, for example, PETSc vector routines) as well as individually.
  One-, two- and three-dimensional distributed arrays (DAs) are used to support structured-grid computations. Their elements can be accessed using global (i.e. with respect to the global grid) or local (with respect to the local on-processor segment of the array) indexing. Provisions are made for overlap zones (ghost points) that can act as buffers for copies of data elements on geometrically neighboring processors. Elements of DAs, like those of distributed vectors, can be set collectively and individually.

With the proper use of assembly routines it is possible, in principle, to program pipeline control structures explicitly, with the advantage that the grouping factor is under user control. However, PETSc allows only one type of distribution for its vectors and DAs, namely blocking (i.e. uni-partitioning). There is no support for more advanced domain decompositions, such as multi-partitioning. Nor is there support for dynamic decompositions, such as those required by transpose-based parallel algorithms. Finally, there are several other data accesses in DAs that are required in complex CFD production codes and that are not supported in PETSc, such as fetches of data from remote processors at points other than ghost points.

3 Charon design philosophy

Based on the survey of projects on parallelization aids for scientific computing in the previous section, and on experience developing advanced parallel programs from scratch, we arrive at the following basic design guidelines for Charon.

3.1 Requirements

1. Every control structure and data access expressible in a serial code should also be expressible in a parallel code. Common control structures should be easy to express;

2. Domain decompositions should be flexible, potentially dynamic, and under complete control of the user. Common domain decompositions should be easy to specify;

3. The user should always be able to get efficient access (i.e. without the need to copy) to memory locations where the (distributed) data is actually stored;

4. Programming in Charon should not have to be done exclusively through explicit function calls;

5. Converting a serial program or design to a parallel program should be easy;

6. Parallel I/O should be straightforward and efficient;

Criteria 1–4 are largely satisfied by the message-passing model, which we adopt for this work. With its wide acceptance and proven efficiency on a large number of platforms, the Message Passing Interface (MPI [45]) is the proper choice. Easy expressibility of common control structures and domain decompositions requires a layer of functions on top of lower-level constructs. Criterion 4 is a corollary to 1; in systems that force the programmer to use library calls alone for accomplishing tasks, functionality is inherently limited, and too much 'foreign language' is required.[2] User control in the user's language (mostly Fortran and C) should be explicitly recognized and supported, not merely allowed. Criteria 5-6 are generally at odds with the low-level functionality and data distribution support of the message-passing model; flexibility and ease of use have been found incompatible in virtually all of the systems surveyed in Section 2. This is because most systems provide only one level of programming

[2]Compare ordering food in a French restaurant; an English speaking customer may be willing to learn a few French words to order a special meal, but will cancel the reservation if the whole dinner conversation has to be conducted in French as well.

support, which also needs to be *efficient* to qualify as a useful instrument. Charon's design includes a hierarchy of control structures, all of which have complete expressibility, but trade ease of use against efficiency.

3.2 Conceptual design

Data distribution and parallel control are orthogonal design features of Charon; it is possible to distribute arrays making up the data structures of a program without touching the sequential logic. i.e. without explicitly parallelizing it. Subsequent code modifications to establish concurrent execution leave the distribution unchanged.

The problem is how to satisfy demand for data that is not local to the calling processor. We will focus on loops over (parts of) multi-dimensional arrays, since this is where most of the work is done in scientific computing. In Charon there are three ways remote data demands can be satisfied. They correspond roughly to the three levels of abstraction mentioned above, and depicted in Figure 1.

1. Implicitly invoked communications.
   Assignments are replaced by calls to the `CHN_assign`, `CHN_address` and `CHN_value` functions (see Section 5), and Charon will make sure that the right values are stored in the correct locations, regardless of data distribution. No code rearrangement or communication calls are needed. It is generally very inefficient, but offers the convenience of serial logic.

2. Structured Charon communications.
   In scientific programs featuring stencil computations, remote data demands often involve logically nearby processors and grid points. Such values can be fetched before a loop is entered and cached in the locations where they are 'expected', i.e. at so-called ghost or overlap points (see Section 4, `CHN_copy(_ghost)_faces_all)`). This obviates the need for the expensive implicitly invoked communications. The ghost points are declared during array setup and are managed by the library, so no separately defined user buffers are needed for copying. The other common structured communication is equivalent to HPF's `redistribute`, and is named similarly.

3. Unstructured Charon communications and MPI function calls.
   When the remote data demands are not of the stencil type, the user will need to fetch the data explicitly and store in and copy from private buffers. This requires a (partial) recoding of the original operations to assimilate the foreign data. Since there is always direct access to distributed data, regular MPI calls may be used for message passing. Alternatively, Charon provides functions for copying Cartesian product subsets of distributed arrays to and from private buffers (`CHN_get/put_tile`; not further described in this paper).

An example of a typical parallel program development process using Charon is shown in Figure 2.

In the course of improving performance, high-level constructs will gradually be replaced by mid- and low-level function calls. As a consequence, top-performing parallel codes derived using Charon will often look very similar to message-passing codes, although many of the

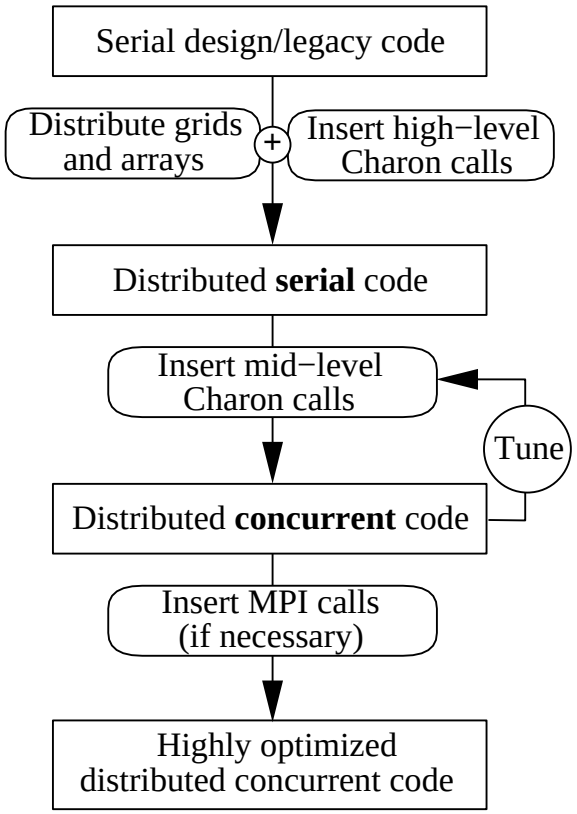

Figure 2: Sample parallelization flowchart

common manipulations will not have to be coded explicitly by the user, but will be provided as tools in the toolkit. The important difference is that Charon codes will have been created in a piecemeal fashion, with support for rapid prototyping and validation. I/O in scientific computing is usually a data-parallel operation, and support for this can be provided easily and transparently using the collective I/O operations defined in the recently announced MPI 2 standard. These operations derive their efficiency from a reshuffle of fragmented data among processors to increase granularity before actually accessing the storage device.

4 Distribution support tools

Charon supports the parallelization of programs using multi-dimensional arrays related to structured grids. The data distribution process consists of three fundamental steps:

1. Define a *grid* and create a partitioning in *cells* using Cartesian sections (see below). The result is a *section* data structure.
2. Assign cells to processors. The result is a *decomposition* data structure.
3. Create the multi-dimensional, distributed array and associate it with a decomposition and local storage space. The result is a *distribution* data structure.

In the following description of Charon functions, the integer variables `grid`, `section`, `decomposition`, and `distribution` (in typewriter font) are handles to the corresponding

data structures. There are Fortran 77 and C bindings for all Charon functions; in this paper we present mainly the C syntax.

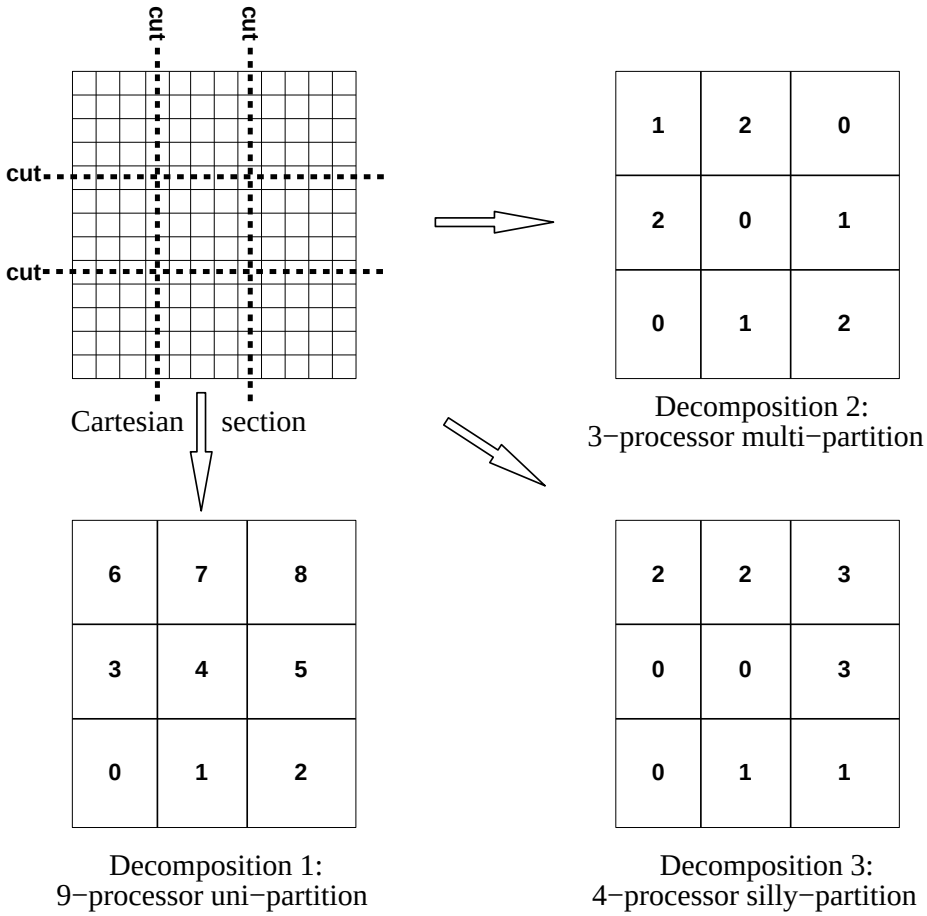

Figure 3: Cartesian section may define several different decompositions. Numbers inside cells indicate processor ownership.

`CHN_create_grid` initializes a discretization grid of a certain dimensionality. `CHN_set_grid_size` and `CHN_set_grid_start` specify the size and starting index of the grid in a particular coordinate direction, respectively. The *grid* data structure and subsequent constructs based on it are defined only for the processors in the MPI [45] communicator specified in `CHN_create_grid` (see page 13).

Structured-grid computations often involve stencil operations that require gathering data from nearby points in the grid. Most useful domain decompositions assign contiguous blocks of points to individual processors, which reduces the amount of communication necessary to fetch nonlocal data. Such domain decompositions are conveniently defined in terms of *Cartesian sections*. These are regular tessellations created by cutting the grid along coordinate planes. In order to retain full flexibility, we separate the construction of Cartesian sections from the assignment of grid blocks or *cells* to individual processors. The result of the assignment process, the *decomposition* data structure, is fundamental to Charon. Decompositions can be tailored to specific needs by modifying them after initialization. Figure 3 shows an example of how a Cartesian section of a two-dimensional grid is used to define

several decompositions. All Charon functions are uniformly applicable to grids of any spatial dimension, although not all are implemented efficiently for dimensionality higher than three.

`CHN_create_section`, `CHN_set_num_cuts`, and `CHN_set_cut` are used to initialize a section based on a grid, to specify the number of cuts in a certain coordinate direction, and to define the value of a particular cut (a cut value of n places a separator between grid points $n-1$ and n), respectively. Alternatively, we may insert all cuts necessary to define a common partitioning scheme in a single library call. The common decompositions currently supported by Charon are uni-partitioning (`CHN_set_unipartition`; each processor is assigned a single cell), diagonal multi-partitioning (`CHN_set_multipartition`; each processor is assigned several cells in a regular pattern [50]), and the degenerate so-called solo-partitioning (`CHN_set_solopartition`; the grid is left undivided, regardless of the number of processors involved). While multi- and solo-partitioning are unambiguous, there are two different, equally reasonable ways to determine a uni-partition decomposition, selected by the `shape` parameter. The shape value `DEFAULT_SHAPE` will try to minimize surface areas of cells, whereas the value `EQUAL_CUTS` will try to insert the same number of cuts in each coordinate direction, regardless of grid aspect ratios. Charon also allows the user to specify that a certain dimension of the grid should *not* be partitioned (`CHN_exclude_partition_dimension`, for uni- and multi-partitioning only).

`CHN_create_decomposition` and `CHN_set_owner` initialize a decomposition based on a certain section and fix ownership of a particular cell, respectively. Ownership is signified by the `rank` of the processor within the MPI communicator specified in `CHN_create_grid`. Because grids can have different dimensionality, the number of indices needed to identify a cell can vary. Whereas the `CHN_set_owner` routine suffices to construct any type of decomposition (see, for example, Decomposition 3 in Figure 3), it is again preferable to assign cells for common decompositions using a single library call (`CHN_set_uni/multipartition_owners`). `CHN_set_solopartition_owners` assigns ownership of all the cells in a decomposition to a single root processor. Before a decomposition can be used, the function `CHN_commit_decomposition` must be called to complete the data structure.
Syntax of grid, section and decomposition functions:

```
int CHN_create_grid(int *grid, MPI_Comm communicator, int num_dims);
int CHN_set_grid_size(int grid, int dir, int size);
int CHN_set_grid_start(int grid, int dir, int start_index);

int CHN_create_section(int *section, int grid);
int CHN_set_num_cuts(int section, int dir, int num_cuts);
int CHN_set_cut(int section, int dir, int cut_num, int cut_value);
int CHN_exclude_partition_dimension(int section, int dir);
int CHN_set_multi/solo/partition(int section);
int CHN_set_unipartition(int section, int shape);

int CHN_create_decomposition(int *decomposition, int section);
int CHN_set_cell_owner(int decomposition, int rank, int cell_num);
int CHN_set_uni/multipartition_owners(int decomposition);
int CHN_set_solopartition_owners(int decomposition, int root);
```

```
int CHN_commit_decomposition(int decomposition);
```

Once a decomposition is formed, a distributed variables (`distribution`) can be associated with it. `CHN_create_distribution` (see below) defines a distributed array of some elementary `data_type` (`MPI_INT`, `MPI_FLOAT`, etc). The user also specifies the tensor `rank` of the variable, plus an array of numbers of `components` for each index of the rank. For example, setting `rank` equal to `2` and `components` equal to `(3,3)` defines a 3×3 matrix at each point of the grid. To accommodate stencil operations the user specifies a nonnegative number of `ghost_points` that form a border of overlap points (communication buffer) around each cell.

`Start_address` refers to a range of memory locations reserved for storage of elements of type `data_type`. In Fortran 77, the starting address is usually the beginning of a user declared array. In C it can be the same, but it may also be NULL, and space can be assigned to the distributed array later through `CHN_set_start_address`. In either case, it is user allocated space that is reserved. The distribution data structure simply provides a structuring interpretation of space pointed to by the user. This makes it possible to operate on distributed data in any way the programmer deems convenient, through Charon functions, through plain Fortran or C, or—most often—both.

By default, all cells take up an equal amount of space. The layout is consistent with a storage declaration that allocates to all cells subarrays of identical dimensions. This makes it easy in both C and Fortran to declare a distributed variable of rank r on a grid of dimension n as an $(n + r + 1)$-dimensional array, where one dimension is of the size of the number of cells owned by each processor.

Complete control over memory usage is got by specifying explicitly where each cell `c` starts in memory, relative to the global starting address (`CHN_set_array_offset`), and what the subarray dimensions are (`CHN_set_array_dimension`).
Syntax of distribution and layout functions:

```
int CHN_create_distribution(int *distribution, int decomposition,
                            MPI_Datatype data_type, void *start_address,
                            int rank, int *component, int ghost_points);
int CHN_set_array_offset(int distribution, int offset, int cell_num);
int CHN_set_array_dimension(int distribution, int dir, int subarray_size,
                            int cell_num);
```

All grid, section, decomposition and distribution creation and manipulation operations are collective, which means that all processors in the corresponding communicator must call these routines with the same parameters (except those that associate storage space on individual processors). In addition to the above assignment and initialization routines, there are also inquiry functions that interrogate the Charon variables, for example to find out the number of a particular cell in the decomposition, or which processor owns a particular grid point. These functions will be described as they are introduced in the examples below.

5 Execution support tools

At the highest level of abstraction, Charon presents a programmer interface that makes the transition from a serial to a *correct* distributed implementation simple and straightforward.

The user need not be concerned about details of the domain decomposition, local and remote data, concurrency, communication, etc. These bookkeeping aspects are hidden. This is generally inefficient, but that is not a problem. In subsequent refinements performance improvements are obtained, again making use of Charon tools.

Charon does not provide automatic code conversion. All parallelization is carried out by the user, who retains complete control over data lay-out and program flow. Hence, the necessary code changes must be kept at a minimum. For that purpose Charon offers execution support tools that simulate a single data space and a single thread of control. Assignments and control structures are exact images of the serial program, and the resulting code is executed by all processors; Charon simulates a single, replicated program counter.

We use the following rationale for the implementation. Each element of a distributed variable is associated with a grid point, which has a unique *owner* processor. So it is most natural—and often least expensive—to employ an *owner-computes* rule: whenever an element of a distributed variable occurs on a left-hand side of an assignment, the processor who owns it is responsible for its update. Since all processors execute the same code within the same control structure, we have to provide a mechanism to skip assignments to *nonlocal* memory locations; the replicated program counter 'pauses' on all processors, except on the one executing the local assignment, and 'resumes' collectively immediately thereafter.

The implementation is as follows. Each assignment in the serial program is converted into a call to `CHN_assign`, which takes as arguments a left hand side (an address) and a right hand side (a value). If the address is `NULL` (not reachable), the assignment is effectively ignored by the calling processor; it is masked. Masking is provided by the function `CHN_address`, which returns an actual location for a *local* element of a distributed variable, and `NULL` otherwise. Assignment masking alone, however, is not sufficient, since assembly of the right hand side of the assignment may involve contributions from local as well as (possibly several) remote memories. For this purpose the generic function `CHN_value` is introduced. It operates on distributed variables and always returns the proper value, regardless of which processor owns the donor point. Analogous to the common *owner-computes* rule for left hand sides of assignments, we apply the *owner-serves* rule for the right hand side components, meaning that the unique processor that owns a particular grid point is responsible for producing values associated with the point upon request by `CHN_value`.

No distinction is made between values returned by the function `CHN_value` and values of nondistributed variables and constants. All are *rvalues* in C terminology. Similarly, no distinction is made between addresses returned by the function `CHN_address`, and addresses of nondistributed variables. Both are *lvalues*. In a correct program using only top-level Charon tools, rvalues always exist, whereas lvalues are only defined if they are local. Alternatively, we may say that the highest level of abstraction of Charon only offers (implicitly invoked) remote gets, not puts.
Syntax of global access functions:

```
int    CHN_assign(void *my_address, <type> my_value);
<type> CHN_value(int distribution, int index1, int index2, ...);
void   *CHN_address(int distribution, int index1, int index2, ...);
```

Note that the `CHN_assign` operator is overloaded; it can take values and addresses of any elementary type, as long as they are consistent (i.e., `my_value` and `my_address` must refer to the

same type). The generic function `CHN_value` specializes to `CHN_float_value`, `CHN_int_value`, etc., depending on the type of the distributed variable. In addition, functions `CHN_value` and `CHN_address` allow variable-length parameter lists so as to accommodate distributed variables of differing dimensions and tensor ranks. This can be done safely in C and Fortran 90, but is not formally allowed in Fortran 77. However, most Fortran 77 compilers will do the right thing, even when issuing warnings. `Index1, index2, ...` are indices with respect to the global grid. Observe that the return value of `CHN_address` is used functionally as an lvalue by Charon, which is not possible in Fortran.

For the above reasons, `CHN_assign`, `CHN_value` and `CHN_address` are all implemented in C. However, they are callable from Fortran. Correctness (i.e. serial consistency) of a program utilizing only these functions is easily shown, even though Charon makes no assumptions about lock-step execution or other synchronization features of the runtime system, and does not pose any restrictions on data dependencies in the program. Each invocation of `CHN_assign` requires the cooperation of the processor that owns a referenced remote data element. Because all processors execute the same code, any update of such referenced remote data occurring logically before the value is requested must already have taken place before the request is registered and satisfied; synchronization is performed automatically. This is equivalent to realigning the replicated program counter.

A side effect of the cooperative nature of implicitly invoked communications is that they must be issued as broadcast operations. A processor executing the `CHN_value` function must take active part in sending data, but cannot know the recipient until `my_address` has been evaluated. Both `my_address` and the expression involving `CHN_value` are arguments of the `CHN_assign` function, and Fortran nor C semantics prescribe a unique evaluation order. Hence, the rvalue may be evaluated before the destination address is known, which implies that the rvalue be available to *all* processors in the communicator. A broadcast is required. If the lvalue is not a distributed variable—in other words, if it is a global variable—the `CHN_address` and `CHN_assign` functions are not used. Global variables are automatically self-consistent, because each processor assigns the same (broadcast) value to its local copy.

`CHN_assign`, `CHN_value`, and `CHN_address` are sufficient, in principle, to move a serial program based on structured grids into a distributed environment while retaining serial logic, and these functions make up the bulk of Charon's top-level distributed execution support. Relaxation of the owner-computes and owner-serves rules and introduction of data-parallel communications are the most important ingredients of mid-level parallel execution support. These will be presented in the examples in the next section.

6 Tuning applications: two examples

Whereas the top level support functions *distribute* a code, they do not yield a truly *parallel* program, since execution is effectively serialized. We will show, by way of two examples based on the NAS Parallel Benchmarks II (NPB) program suite [4], how to obtain high performance in a gradual fashion.

6.1 Tri-diagonal ADI

The first example is for illustrative purposes only. It is based on the NPB Block Tri-diagonal (BT) problem [3], which defines an ADI (Alternating Direction Implicit) algorithm to solve

a system of nonlinear partial differential equations on a 3D grid. We reduce the system to a simple linear scalar equation on a 2D grid:

$$\frac{\partial q}{\partial t} = \frac{\partial q^2}{\partial x^2} + \frac{\partial q^2}{\partial y^2} \tag{1}$$

The interesting part of the ADI algorithm concerns the inversion of sets of banded tridiagonal matrices (so-called *factors*), since they involve recurrences in different coordinate directions. There is one factor for each grid line, in each of the two coordinate directions. Here is the C pseudo code for a single time step of the ADI algorithm.

```
for (each grid point) compute_residual(rhs,q);         /* rhs: residual    */
for (dir=0; dir<2; dir++) invert_factor(lhs,rhs,dir); /* lhs: banded matx */
for (each grid point) update(q,rhs);                   /* q = q + rhs      */
```

From now on we will use indices `i` and `j` for discrete coordinates in the x- and y-directions, respectively. Updating the solution `q` is easiest, since it is an operation that is data parallel, and that involves no difference operators. Assuming a square grid of size $\mathtt{N}^2$ points (indexing starts at 1), the serial update function

```
for (i=1; i<=N; i++) for (j=1; j<=N; j++) q[i][j] += rhs[i][j];
```

is converted into:

```
for (i=1; i<=N; i++) for (j=1; j<=N; j++)
  CHN_assign(CHN_address(q_,i,j),CHN_value(q_,i,j)+CHN_value(rhs_,i,j));
```

Here the array `rhs`, which originally contained the residual, has been overwritten with the solution update vector. In the sequel we will use the notation `X_` for the handle to the distributed array `X`. It is based on the grid decomposition (produced by `CHN_create_decomposition`) whose handle is `dcmp`. For brevity we will only use the generic names of Charon access functions. In C all Charon functions invoked for their side effects return an integer error value, which can be ignored by the programmer. In Fortran such error values are returned through the parameter list, and hence require an additional parameter.

Notice that the structure of the distributed loop nest is identical to that of the original version, and that we have made no assumptions about how the grid has been partitioned. The loop nest is completely serialized; only one processor makes assignments to the distributed variable `q` at any one time.

A simple optimization is to bypass execution of `CHN_assign` statements involving remote data elements, so that no broadcasts are needed. This requires relaxing the principle of the replicated program counter. Charon demarcates an environment within which the program counters are allowed to float by the bracketing statements `CHN_begin_local`, `CHN_end_local`. Within the environment broadcasts are suppressed The original structure of the loop nest is retained by using `CHN_point_owner` to test whether a point in the grid is assigned to the calling processor (identified by integer `my_rank`, supplied by MPI). We only execute the loop body if the outcome is true. The following code eliminates all redundant synchronizations and communications:

```
CHN_begin_local(dcmp);
for (i=1; i<=N; i++) for (j=1; j<=N; j++)
 if (CHN_point_owner(dcmp,i,j)==my_rank)
     CHN_assign(CHN_address(q_,i,j), CHN_value(q_,i,j) +
                                     CHN_value(rhs_,i,j));
CHN_end_local(dcmp);
```

This version of the loop nest is still much more expensive than a hand-coded message-passing version, since function calls are made during every iteration. Moreover, all processors do need to perform the ownership test for all elements of the iteration index space. The next optimization is obtained by restricting the iteration index space *a priori* to cells owned by the calling processor. The inquiry functions `CHN_own_total_num_cells(dcmp)` and `CHN_global_cell_index(dcmp,c)` return the total number of cells owned by the calling processor, and the sequence number of the `c`th cell owned by the calling processor within the overall decomposition, respectively. `CHN_cell_start/end(dcmp,dir,c)` returns the start/end grid indices of cell `c` in coordinate direction `dir`.

```
CHN_begin_local(dcmp);
for (c=0; c<CHN_own_total_num_cells(dcmp); c++) {
 cg = CHN_global_cell_index(dcmp,c);
 for (i=CHN_cell_start(dcmp,0,cg); i<=CHN_cell_end(dcmp,0,cg); i++)
 for (j=CHN_cell_start(dcmp,1,cg); j<=CHN_cell_end(dcmp,1,cg); j++)
  CHN_assign(CHN_address(q_,i,j,k,m),CHN_value(q_,i,j,k,m) +
                                     CHN_value(rhs_,i,j,k,m));
}
CHN_end_local(dcmp);
```

The above code modification has two important implications. The first is that now the domain decomposition is exposed, although the loop is still valid for any possible decomposition. The second is that the original loop has been reordered. However, due to the lack of data dependences across iterations of the loop, this reordering is safe.

Notice that point indexing is still with respect to the global grid. The price for this convenience is the calls to `CHN_assign`, `CHN_address` and `CHN_value`. We now eliminate these and write the complete final loop as[3]:

```
for (c=0; c<CHN_own_total_num_cells(dcmp); c++) {
 cg = CHN_global_cell_index(dcmp,c);
 for (i=1; i<=CHN_cell_size(dcmp,0,cg); i++)
 for (j=1; j<=CHN_cell_size(dcmp,1,cg); j++)
  q[c][i][j] += rhs[c][i][j];
}
```

[3]Note that `q` and `rhs` have been redimensioned to accommodate the extra cell index `c`. The multi-index C arrays in the example are printed only for clarity. In our (and most other) real applications, variable-size multi-dimensional arrays in C are implemented using macros that compute offsets into one-dimensional arrays.

CHN_cell_size is defined as CHN_cell_end − CHN_cell_start + 1. Now there is no more need for the CHN_begin/end_local pair, because there are no calls to the CHN_value function. We have finally obtained a code fragment that is as efficient as a hand-coded message-passing version. It is also almost as complicated as that, so it would appear that nothing has been gained. However, the important difference with other systems is that this optimized code may be freely combined with high-level unoptimized code fragments that do not contain any explicit references to the domain decomposition. Moreover, the bookkeeping of the decomposition does not have to be done by the user.

We now move to the more complex example of operations on arrays involving recurrences. The serial code represents the forward-elimination phase of the inversion of the x-factor in our sample ADI program.

```
for (i=1; i<=N-1; i++) for (j=1; j<=N; j++) {
 inv             = 1.0/lhs[1][i][j];    /* compute pivot reciprocal */
 lhs[2][i][j]   *= inv;                          /* scale matrix row */
 rhs[i][j]      *= inv;                          /*   "      "    "  */
 lhs[1][i+1][j] -= lhs[2][i][j]*lhs[0][i+1][j]; /* update next row  */
 rhs[i+1][j]    -= rhs[i][j]   *lhs[0][i+1][j]; /*   "      "   "   */
}
```

Here the array lhs represents a family of banded, tri-diagonal matrices, one for each grid line in the x-direction (i.e. one for each value of j). The first index selects the particular band of the matrix. Index 0 corresponds to the lower, 1 to the main, and 2 to the upper diagonal. We again convert using only top-level Charon functions:

```
for (i=1; i<=N-1; i++) for (j=1; j<=N; j++) {
 inv = 1.0/CHN_value(lhs_,1,i,j);
 CHN_assign(CHN_address(lhs_,2,i,j),  inv*CHN_value(lhs_,2,i,j));
 CHN_assign(CHN_address(rhs_,i,j),    inv*CHN_value(rhs_,i,j));
 CHN_assign(CHN_address(lhs_,1,i+1,j),CHN_value(lhs_,1,i+1,j) -
                       CHN_value(lhs_,2,i,j)*CHN_value(lhs_,0,i+1,j));
 CHN_assign(CHN_address(rhs_,i+1,j),  CHN_value(rhs_,i+1,j)   -
                       CHN_value(rhs_,i,j)  *CHN_value(lhs_,0,i+1,j));
}
```

Notice again that in the transformed code fragment no influence of the domain decomposition is visible. The situation changes when we start to optimize. First assume that the grid is partitioned in a slicewise fashion, such that all points on grid lines in the i-direction are within the same cell, and that each processor receives one such slice (uni-partition scheme). The tri-diagonal systems can be solved independently by all processors, without any communication. Hence, the first optimization is again obtained by using the CHN_begin/end_local pair. Skipping a few steps, we easily arrive at the following efficient code:

```
cg = CHN_global_cell_index(dcmp,0);
for (i=1; i<=N-1; i++) for (j=1; j<=CHN_cell_size(dcmp,0,cg); j++) {
 inv            = 1.0/lhs[1][i][j];
 lhs[2][i][j]   *= inv;
```

```
 rhs[i][j]       *= inv;
 lhs[1][i+1][j] -= lhs[2][i][j]*lhs[0][i+1][j];
 rhs[i+1][j]    -= rhs[i][j]   *lhs[0][i+1][j];
}
```

The problem becomes more interesting when the grid is partitioned differently, for example because there are conflicting recurrences in the program (e.g. in the y-factor). Now two approaches are available.

The first is similar to that offered by HPF, namely the `CHN_redistribution` facility. Before the iterations start, two different decompositions are defined that are aligned with the x- and y-grid lines, respectively. The corresponding distributions are `rhsx` and `lhsx`, and `rhsy` and `lhsy`, respectively. Switching from x-aligned to y-aligned `lhs` distributions, for example, is established by calling `CHN_redistribute(lhsy,lhsx)`.

The second approach leaves the data distributions intact. We choose the multi-partition domain decomposition (see, for example, Figure 3, Decomposition 2), which has the special property that each processor owns a cell in each row and each column of cells of the grid. Hence, if the solution process advances by rows or columns of cells in the respective coordinate directions—so as to respect the recurrence relations in these directions—, a perfect load balance ensues. We note that none of the systems surveyed in Section 2 has the flexibility of supporting multi-partitioning, but in Charon it is easily defined.

For the inversion of the x-factor we first force the solution algorithm to proceed along columns of cells, but retain the convenience of implicitly invoked remote gets. The function `CHN_num_cells(dcmp,dir)` is used to return the number of cells in the decomposition in coordinate direction `dir`.

```
for (ip=0; ip<CHN_num_cells(dcmp,0); ip++)
for (jp=0; jp<CHN_num_cells(dcmp,1); jp++) {
 c = CHN_cell_index(dcmp,ip,jp);
 for (i=CHN_cell_start(dcmp,0,c); i<=min(N-1,CHN_cell_end(dcmp,0,c); i++)
 for (j=CHN_cell_start(dcmp,1,c); j<=CHN_cell_end(dcmp,1,c); j++) {
  inv = 1.0/CHN_value(lhs_,1,i,j);
  CHN_assign(CHN_address(lhs_,2,i,j),   inv*CHN_value(lhs_,2,i,j));
  CHN_assign(CHN_address(rhs_,i,j),     inv*CHN_value(rhs_,i,j));
  CHN_assign(CHN_address(lhs_,1,i+1,j), CHN_value(lhs_,1,i+1,j)-
                          CHN_value(lhs_,2,i,j)*CHN_value(lhs_,0,i+1,j));
  CHN_assign(CHN_address(rhs_,i+1,j),   CHN_value(rhs_,i+1,j)-
                          CHN_value(rhs_,i,j)*  CHN_value(lhs_,0,i+1,j));
  }
}
```

Note that the original loop has been reordered, but that the recurrence relation is respected. Next, the need for frequent implicitly invoked communications must be removed. In keeping with Charon's design philosophy (see page 10, item 2), we try to satisfy remote data demands by aggregated prefetching into the 'expected' locations (i.e. ghost points) using structured communications. As the recurrence has a depth of only one (the half width of the

discretization stencil), a border of ghost points of size one around each cell suffices. Copying ghost point values from neighboring cells is done by function `CHN_copy_faces`, which is described in the next section. Using values stored at ghost points requires relaxation of the owner-serves rule. Similarly, writing values into ghost point locations requires relaxation of the owner-computes rule. The mechanism for causing both rules to be relaxed is the bracketing pair `CHN_begin/end_ghost_access`, which takes as arguments the decomposition `dcmp`, and the particular cell `c` whose ghost points should be made accessible. It is as if temporarily cell `c` has been enlarged by its ghost points.

Before the loop nest is entered, we copy `lhs` and `rhs` values immediately 'ahead' of each column of cells into the ghost point locations (see, for example, Figure 4, bottom right tableau). Thus, when the loop body is executed for the last column of points in each column of cells, all its remote data requirements are automatically satisfied. After all `rhs` and `lhs` ghost point values are written for a whole column of cells, the updated values are transferred in bulk to the next column using function `CHN_copy_ghost_faces`, also explained in Section 6.1.1. The following program results:

```
CHN_copy_faces(rhs_,NONPERIODIC,1,0,LEFT,ALL,ALLVEC,ALLVEC);
CHN_copy_faces(lhs_,NONPERIODIC,1,0,LEFT,ALL,ALLVEC,ALLVEC);
for (ip=0; ip<CHN_num_cells(dcmp,0); ip++) {
  for (jp=0; jp<CHN_num_cells(dcmp,1); jp++) {
    c = CHN_cell_index(dcmp,ip,jp);
    CHN_begin_ghost_access(dcmp,c);
    for (i=CHN_cell_start(dcmp,0,c); i<=min(N-1,CHN_cell_end(dcmp,0,c); i++)
    for (j=CHN_cell_start(dcmp,1,c); j<=CHN_cell_end(dcmp,1,c; j++) {
      inv = 1.0/CHN_value(lhs_,1,i,j);
      CHN_assign(CHN_address(lhs_,2,i,j),   inv*CHN_value(lhs_,2,i,j));
      CHN_assign(CHN_address(rhs_,i,j),     inv*CHN_value(rhs_,i,j));
      CHN_assign(CHN_address(lhs_,1,i+1,j), CHN_value(lhs_,1,i+1,j)-
                           CHN_value(lhs_,2,i,j)*CHN_value(lhs_,0,i+1,j));
      CHN_assign(CHN_address(rhs_,i+1,j),   CHN_value(rhs_,i+1,j)-
                           CHN_value(rhs_,i,j)*  CHN_value(lhs_,0,i+1,j));
    }
    CHN_end_ghost_access(dcmp,c);
  }
  CHN_copy_ghost_faces(lhs_,NONPERIODIC,1,0,RIGHT,ip,ALLVEC,ALLVEC);
  CHN_copy_ghost_faces(rhs_,NONPERIODIC,1,0,RIGHT,ip,ALLVEC,ALLVEC);
}
```

Due to the prefetching, this loop structure no longer requires the implicitly invoked communications. Consequently, we can relax the principle of the simulated single program counter and let each processor execute only its own part of the loop, using the inquiry function `CHN_cell_owner` and the `CHN_begin_/end_local` pair.

The resulting program produces very good performance for a multi-partition decomposition (see Section 6.2), but suffers from severe load imbalance when applied to a uni-partition decomposition. The interested reader is referred to [51] for an explanation of how to pipeline the above code for improved load balance on uni-partition decompositions.

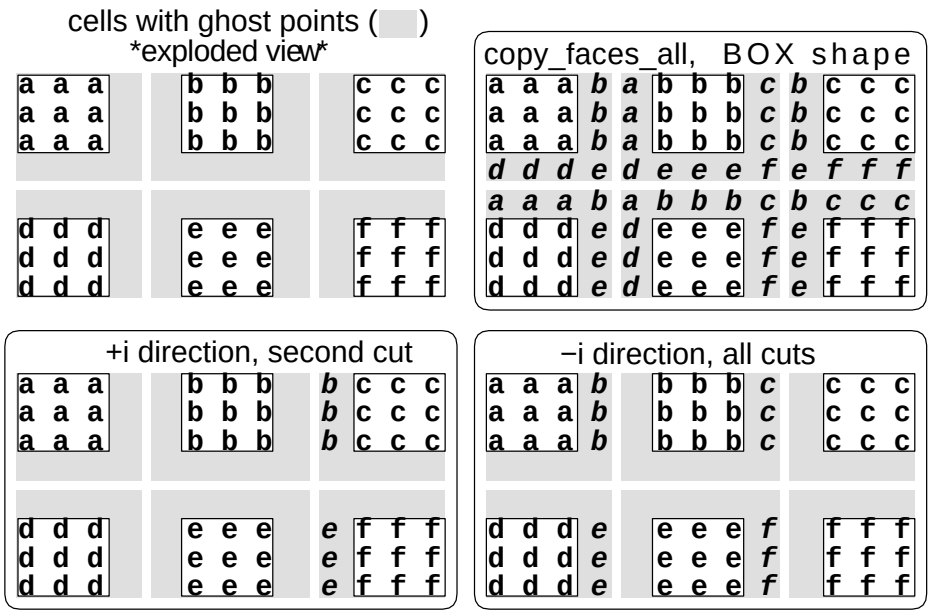

Figure 4: Applications of copy_faces(_all) to a distributed array.

6.1.1 Copying ghost point values

Syntax of face copy functions:

```
int CHN_copy_(ghost_)faces(int distribution, int periodicity, int thickness,
                           int dir, int side, int cut, int *start, int *end);
int CHN_copy_(ghost_)faces_all(int distribution, int periodicity,
                               int thickness, int shape);
```

`CHN_copy_faces` copies values from the outermost 'layers' of cells to the corresponding ghost points of adjacent cells. `Thickness` refers to the number of layers to be copied. Copying takes place to neighboring cells in the `dir`th coordinate direction. The value of `side` determines if the positive (`RIGHT`) or negative (`LEFT`) coordinate direction is chosen, or both (`BOTH_SIDES`). `Cut` specifies the sequence number of the cut in the coordinate direction defined by `dir` across which the copy operation is to take place. Setting `cut` equal to `ALL` selects all cuts simultaneously (if `periodicity` is set to `PERIODIC`, copying wraps around the grid). Additionally, we may restrict the copying to a subset of interface points. Vectors `start` and `end` indicate, for each coordinate direction other than that normal to the face to be copied, the start and end coordinates of the points of that subset with respect to the global grid. Alternatively, use `ALL` for the starting index if all points along the cut are to be copied. In the code fragment above we predefine the useful vector `ALLVEC[2]` to be {`ALL,ALL`}.

Often it will be useful to copy face values at all cuts in all directions, especially in the case of explicit methods, where all off-cell information can be fetched beforehand. For this purpose `CHN_copy_faces_all` is provided. It takes as an argument the stencil `shape`, which can have the values `BOX` or `STAR` (see e.g. [5]). The `STAR` shape ignores diagonal values and only copies between strongly connected cells. `BOX` also copies values to weakly connected

cells. Figure 4 shows several different applications of the `CHN_copy_faces(_all)` routines. `CHN_copy_ghost_faces` is similar to `CHN_copy_faces`, but copies values *from* ghost points *to* points properly owned by neighboring cells.

Both types of copy functions are data-parallel, in principle; all processors may participate, but only operate on the data that they own. If a processor is responsible for neither sending nor receiving, it can safely skip the copy call. The result of copy operations is independent of the number of processors involved, but depends only on the number and lay-out of the cells. This is true for all Charon operations defined so far; distributed programs can be simulated, tested and debugged, to a large extent, while using only a single processor. One may even use Charon exclusively to obtain blocked uniprocessor code that optimizes data locality.

Systems like KeLP and PETSc also provide copy utilities, but these are limited to the equivalents of `CHN_copy_faces_all`, and hence do not support data dependencies of the type described in Section 6.1.

6.2 Scalar Penta-diagonal NAS Parallel Benchmark

We use the techniques described in Section 6.1 to distribute and parallelize the serial version of the Scalar Penta-diagonal (SP) NAS Parallel Benchmark program (Fortran 77). The code is significantly more complex than the simple example of the scalar tri-diagonal ADI problem, and some preparation is needed to apply Charon conveniently.

First, we define for all 3D arrays two distributions. One is associated with the multi-partition decomposition, and the other is associated with the degenerate solo-decomposition, consisting of a single cell of the size of the whole grid. Subsequently, we compose two simple subroutines (named `to_solo` and `to_distributed`), which map all distributed arrays to the single cell on one processor, and vice versa. Mapping between distributions takes place with a single call to `CHN_redistribute`. The significance of the two mapping routines is that they allow us to distribute and parallelize a single subroutine, loop, or even statement, while keeping all the rest of the code *completely unchanged*. Two serial segments of the code can be connected by the following sequence:

```
call to_distributed
distributed code ( either Charon or otherwise)
call to_solo
```

This is especially important when the code is large, or when the invocation of top-level Charon functions would generate so much communication that distributing the whole program at once would lead to unacceptably long run times.

A second convenient device is the definition of customized functions that are abbreviations for Charon operations. For example, we may define the following functions (`lhs_` is assumed contained in the include file `my_Charon_pars.h`):

```
subroutine lhsassign(i,j,k,m, value)
include 'my_Charon_pars.h'
integer i,j,k,m, ignore_error
double precision value
call CHN_assign(CHN_address(lhs_,i,j,k,m),value, ignore_error)
return
```

```
  end

  double precision function lhsval(i,j,k,m)
  include 'my_Charon_pars.h'
  integer i,j,k,m
  lhsval = CHN_double_precision_value(lhs_,i,j,k,m)
  return
  end
```

This allows us to convert the serial statement

```
  lhs(i1,j,k,n+3) = lhs(i1,j,k,n+3) -
 >                  lhs(i1,j,k,n+2)*lhs(i,j,k,n+4)
```

into the relatively terse

```
  call lhsassign(i1,j,k,n+3,lhsval(i1,j,k,n+3) -
 >                          lhsval(i1,j,k,n+2)*lhsval(i,j,k,n+4))
```

Finally, it is also convenient to store certain frequently used parameters such as the begin and end indices of cells, instead of using the full names of Charon access functions each time.

With these preparations the parallelization of SP from the serial code distributed by the NAS Parallel Benchmark Group was a fairly simple process, which moved in small steps while maintaining program correctness all the time. The converted codes grew from an original 3072 lines to 3663 lines, which is a modest increase considering the logical complexity of the multi-partition decomposition.

Explicit Charon communications were necessary only in four routines, namely that used for computing the residuals at the beginning of each time step (`CHN_copy_faces_all`, and those used to invert the three factors (`CHN_copy(_ghost)_faces`). Since SP solves a system of partial differential equations, the `lhs` and `rhs` arrays are more complex than those in Section 6.1; they have 15 and 5 values per grid point, respectively. Moreover, SP inverts penta-diagonal matrices, as opposed to the tri-diagonal matrices in Section 6.1, so ghost point borders of thickness two are required to accommodate prefetching.

Per cell interface point and per factor a total of 90 double precision values are copied, namely two rows of `lhs` and `rhs` before entering the forward elimination loop nest of the factor inversion ($2 * 15 + 2 * 5 = 40$ values), two rows of each during the advancement of the forward elimination (40 values), and two rows of `rhs` (10 values) during the advancement of the backsubstitution. By comparison, the hand-coded MPI version provided by NAS copies only 32 values per interface point. This significant difference is due to two reasons. The first is that the present version of Charon only allows copying of *all* tensor components of a distributed array, whereas only the upper diagonal element of `lhs` need to be passed to the next layer of cells in the forward elimination phase. The second reason is that the technique described in Section 6.1 insists on executing the assignments from the serial code in the same order for all points of each cell owned by each processor. But if we change the code only very little, namely by letting each loop over a cell start at the first ghost point instead of at the first properly owned point of the cell, we can avoid the advance copying of `lhs` and `rhs` before entering the loop nest altogether, thus saving 40 double precision values per

Table 1: Performance (total Mflops/s) of NPB SP code on Origin 2000, parallelized using Charon

	Standard comm.		Modified comm.	
# procs	Mflops/s	speedup	Mflops/s	speedup
1	61.2	1	61.2	1
4	172.2	2.81	186.9	3.05
16	713.9	11.5	916.3	15.0
64	1952.	31.9	2580.	42.2

interface point. This technique requires some duplication of computation among successive cells, but, as is often the case in parallel computing, this trade-off between computation and communication pays off significantly.

In Table 1 we show the results of running the SP code on 1, 4, 16, and 64 processors of an SGI Origin 2000 system with 250MHz nodes running Irix 6.5 SE MR. The column named 'Modified comm.' shows the effect of reducing the communication cost to 50 values per interface points, compared to the 90 interface values for 'Standard comm.'

The disproportionally large performance increase when going from 4 to 16 processors is most likely due to an improvement in cache performance caused by the decreasing size of the data set per processor. It follows from the table that with relatively little effort a rather well-performing parallel version of a non-trivial program can be obtained. Further tuning is possible, for example by storing upper diagonal elements of `lhs` in dedicated arrays just before the face copy, so that only the minimum amount of data is communicated.

7 Conclusions and future work

We have excerpted the Charon toolkit for incremental parallelization of scientific programs based on structured grids. The library allows for quick and convenient development of parallel code, and provides a mechanism to continually check correctness of the program under development. Tools are available to customize data distributions, but the most common useful distributions are preprogrammed. The normal progression of parallel program development through Charon involves a gradual replacement of relatively inefficient high-level calls with fast-performing lower level constructs, possibly including explicit message passing. However, high and low level calls can always be mixed, which is especially useful when adding new features with complicated data dependencies to a previously tuned code. Charon programs tend to be shorter and easier to read and maintain than traditional message-passing programs, due to the encapsulation of domain decomposition details. Nonetheless, the library allows the user complete control over memory usage and lay-out. Performance of the library for the NAS Parallel Benchmark SP code is good.

The two main disadvantages of Charon are that some hand-coding is required, and that the user must have some knowledge of the program structure at hand.

Planned future extensions are a facility for copying only parts of the tensor components of distributed arrays, the implementation of a parallel I/O function, and facilities for generalized distributions that support legacy code techniques like overindexing (common when creating long loops for vector machines) and lower-dimensional work arrays and index swaps (common when attempting to improve data locality).

References

[1] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, Y.J. Wu, "PLAPACK: Parallel linear algebra package," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," IEEE Computer, Vol. 29, pp. 18–28, February 1996

[3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, S.K. Weeratunga, "The NAS Parallel Benchmarks," Int. J. Supercomputing Appl. Vol. 5, pp. 63-73, 1991

[4] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, M. Yarrow, "The NAS parallel benchmarks 2.0," Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, December 1995

[5] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," Modern Software Tools in Scientific Computing, E. Arge, A.M. Bruaset, H.P. Langtangen, Ed., Birkhauser Press, 1997

[6] S. Balay, W.D. Gropp, L. Curfman McInnes, B.F. Smith, "PETSc 2.0 Users manual," Report ANL–95/11 - Revision 2.0.17, Argonne National Laboratory, Argonne, IL, 1997

[7] L.S. Blackford et al., "ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance," Supercomputing '96, Pittsburgh, PA, November 1996

[8] P. Boulet, T. Brandes, "Evaluation of automatic parallelization strategies for HPF compilers," Research Report 95-44, Ecole Normale Supérieure de Lyon, France, November 1995

[9] D.L. Brown, G.S. Chesshire, W.D. Henshaw, D.J. Quinlan, "Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997

[10] D.C. Cann, "The Optimizing SISAL Compiler: Version 12.0," Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA 94550, April 1992

[11] N. Carriero, D. Gelernter, "How to write parallel programs: a guide to the perplexed," ACM Computing Surveys, Vol. 21, pp. 323–357, 1989

[12] J.B. Carter, J.K. Bennett, W. Zwaenepoel, "Techniques for reducing consistency-related communication in distributed shared memory systems," ACM Trans. Computer Syst., Vol. 13, pp. 205–243, August 1995

[13] B. Chapman, P. Mehrotra, J. Van Rosendale, H. Zima, "A software architecture for multi-disciplinary applications: Integrating task and data parallelism," Technical Report 94-18, ICASE, NASA Langley Research Center, Hampton, VA, March 1994

[14] B. Chapman, P. Mehrotra, H. Zima, "Programming in Vienna Fortran," Scientific Programming, Vol. 1, pp. 31–50, 1992

[15] S.W. Chappelow, P.J. Hatcher, J.R. Mason, *Optimizing data-parallel stencil computations in a portable framework*, 3^{rd} Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Troy, NY, May 1995

[16] C. Clémençon, K.M. Decker, V.R. Deshpande, A. Endo, J. Fritscher, P.A.R. Lorenzo, N. Masuda, A. Müller, R. Rühl, W. Sawyer, B.J.N. Wylie. F. Zimmerman, "Tools-supported HPF and MPI parallelization of the NAS Parallel Benchmarks," 6^{th} Symp. Frontiers of Massively Parallel Computing (Frontiers'96, Annapolis), pp. 309-318, IEEE Comp. Soc. Press, 1996

[17] M. Colajanni, M. Cermele, "DAME: An environment for preserving efficiency of data parallel computations on distributed systems," IEEE Concurrency, pp. 41-55, January-March 1997

[18] Culler et al., "Parallel programming in Split-C," Supercomputing '93, Portland, OR, pp. 262–273, November 1993

[19] S.J. Fink, S.B. Baden, S.R. Kohn, "Flexible communication mechanisms for dynamic structured applications," 3^{rd} Int. Workshop Parallel Algorithms for Irregularly Structured Problems, Santa Barbara, CA, August 1996

[20] I.T. Foster, K.M. Chandy, "Fortran M: A language for modular programming," Technical Report MCS-P327-0992, Argonne National Laboratory, Argonne, IL, June 1992

[21] I.T. Foster, D.R. Kohr, R. Krishnaiyer, A. Choudhary, "Double standards: Bringing task parallelism to HPF via the Message Passing Interface," Supercomputing '96, Pittsburgh, PA, November 1996

[22] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, M. Wu, "Fortran D language specification," Technical Report TR90079, Dept. of Computer Science, Rice University, December 1990

[23] Gannon et al., "Implementing a parallel C++ runtime system for scalable parallel systems," Supercomputing '93, Portland, OR, pp. 588–597, November 1993

[24] A.S. Grimshaw, "Easy to use object-oriented parallel programming with Mentat," IEEE Computer, pp. 39-51, May 1993

[25] T. Gross, D. O'Hallaron, J. Subhlok, "Task parallelism in a High Performance Fortran framework," IEEE Parallel & Distr. Technology, Vol. 2, pp. 16-26, 1994

[26] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S-W. Liao, M.S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," Supercomputing '95, San Diego, CA, December 1995

[27] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," IEEE Computer, Vol. 29, pp. 84–89, December 1996.

[28] S. Hiranandani, K. Kennedy, C. Tseng, "Preliminary experiences with the Fortran D compiler," Supercomputing '93, Portland, OR, pp. 338–350, November 1993

[29] E.N. Houstis, S.B. Kim, S. Markus, P. Wu, A.C. Catlin, S. Weerawarana, T.S. Papatheodorou, *Parallel ELLPACK Elliptic PDE Solvers*, INTEL SuperComputer User's Group Conf., Albuquerque, NM, June 1995.

[30] S.A. Hutchinson, J.N. Shadid, R.S. Tuminaro, "Aztec User's Guide: Version 1.1," Sandia National Laboratories Technical Report SAND95-1559, Albuquerque, NM, October 1995

[31] C.S. Ierotheou, S.P. Johnson, M. Cross, P.F. Leggett, "Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes," Parallel Computing, Vol. 22, pp. 163–195, 1996

[32] E. Johnson, D. Gannon, "Programming with the HPC++ parallel Standard Template Library," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997

[33] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, M. Zosel, "The High Performance Fortran Handbook," MIT Press, Cambridge, MA, 1994

[34] A. Krommer, M. Derakhshan, S. Hammarling, "Solving PDE problems on parallel and distributed computer systems using the NAG parallel library," High Performance Computing and Networking '97, Vienna, Austria, April 1997

[35] V. Kumar, A. Grama, A. Gupta, G. Karypis, "Introduction to parallel computing," Benjamin/Cummings, 1994

[36] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," IEEE Trans. Computers, Vol. 28, pp. 690–691, 1979

[37] K. Li, P. Hudak, "Memory coherence in shared virtual memory systems," ACM Trans. Computer Syst., Vol. 7, pp. 321-359, November 1989

[38] T. Ngo, L. Snyder, B. Chamberlain, "Portable performance of data parallel languages," Supercomputing '97, San Jose, CA, November 1997

[39] J. Nieplocha, R.J. Harrison, R.J. Littlefield, "The global array programming model for high performance scientific computing," SIAM News, Vol. 28, August-September 1995

[40] R. Ponnusamy, Y-S. Hwang, R. Das, J. Saltz, A. Choudhary, G. Fox, "Supporting irregular distributions in Fortran D/HPF compilers," Technical report CS-TR-3268, University of Maryland, Department of Computer Science, 1994

[41] Y. Saad, S. Kuznetsov, G-C. Lo, A. Malevsky, A. Chapman, "PSPARSLIB: A portable library of parallel sparse iterative solvers," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997

[42] D.J. Scales, K. Gharachorloo, "Towards transparent and efficient software distributed share memory," 16^{th} ACM Symp. Operating Syst. Principles, Saint-Malo, France, October 1997

[43] S.J. Scherr, "Implementation of an explicit Navier-Stokes algorithm on a distributed memory parallel computer," AIAA Paper 93-0063, 31^{st} Aerospace Sciences Meeting and Exhibit, Reno, NV, 1993

[44] W. Schönauer, H. Häfner, R. Weiss, "LINSOL, a parallel iterative linear solver package of generalized CG-type for sparse matrices," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, March 1997

[45] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra, "MPI: The Complete Reference," MIT Press, 1995

[46] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, M. Scott, "Cashmere-2L: Software coherent shared memory on a clustered remote-write network," 16^{th} ACM Symp. Operating Syst. Principles, Saint-Malo, France, October 1997

[47] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, "PARTI primitives for unstructured and block structured problems," Computing Systems in Engineering (Noor's Flight Systems Conference), Vol. 3, pp 73-86, 1992

[48] J.A. Turner, R.C. Ferrel, D.B. Kothe, "JTPack90: A parallel, object-based, Fortran90 linear algebra package," 8^{th} SIAM Conf. Parallel Proc. Scientific Computing, Minneapolis, MN, March 1997

[49] R.F. Van der Wijngaart, "Efficient implementation of a 3-dimensional ADI method on the iPSC/860," Supercomputing '93, Portland, OR, November 1993

[50] R.F. Van der Wijngaart, M. Yarrow, M.H. Smith, " An architecture-independent parallel implicit flow solver with efficient I/O," 8^{th} SIAM Conf. Parallel Proc. for Scientific Computing, Minneapolis, MN, Mar 1997

[51] R.F. Van der Wijngaart, "Charon message passing toolkit for scientific computations," NAS Technical Report, to appear Summer '98, NASA Ames Research Center, Moffett Field, CA

[52] ——, "Cilk-5.1 (Beta 1) reference manual," MIT Laboratory for Computer Science, Cambridge, MA, November 1997

[53] ——, "CM Fortran Reference Manual, version 5.2," Thinking Machines Corporation, Cambridge, MA, 1989

[54] ——, "Forge Explorer User's Guide," Advanced Parallel Research

[55] ——, "CF90 commands and directives reference manual, SR-3901.10," Cray Research, Inc., 1993